International® Baccalaureate

Extended essay cover

Candidates must complete this page and then give this cover and their final version of the extended essay to their supervisor.			
Candidate session number			
Candidate name			
School number			
School name			
Examination session (May or November)	MAY	Year	2013

Diploma Programme subject in which this extended essay is registered: COMPUTER SCIENCE

(For an extended essay in the area of languages, state the language and whether it is group 1 or group 2.)

Title of the extended essay: SHOULD COMPUTER PROCESSOR DEVELOPMENT FOCUS BE SHIFTED FROM SEQUENTIAL TO PARALLEL COMPUTATION?

Candidate's declaration

This declaration must be signed by the candidate; otherwise a grade may not be issued.

The extended essay I am submitting is my own work (apart from guidance allowed by the International Baccalaureate).

I have acknowledged each use of the words, graphics or ideas of another person, whether written, oral or visual.

I am aware that the word limit for all extended essays is 4000 words and that examiners are not required to read beyond this limit.

This is the final version of my extended essay.

Candidate's signature: Date:

Supervisor's report and declaration

The supervisor must complete this report, sign the declaration and then give the final version of the extended essay, with this cover attached, to the Diploma Programme coordinator.

Name of supervisor (CAPITAL letters)

Please comment, as appropriate, on the candidate's performance, the context in which the candidate undertook the research for the extended essay, any difficulties encountered and how these were overcome (see page 13 of the extended essay guide). The concluding interview (viva voce) may provide useful information. These comments can help the examiner award a level for criterion K (holistic judgment). Do not comment on any adverse personal circumstances that may have affected the candidate. If the amount of time spent with the candidate was zero, you must explain this, in particular how it was then possible to authenticate the essay as the candidate's own work. You may attach an additional sheet if there is insufficient space here.

This paper is an excellent description of parallelism in computer programming. I am especially impressed with the Java program, written to accompany the paper and support his conclusions about parallelism. He used some contemporary writers such as Kaminsky as the foundation for his ideas and cited several older works on the subject.

The candidate did his research as a student in my IB Computer Science class, using some books from my own collection and computers in my classroom. He was able to discuss the subject with me and at least one other computer science student.

I believe the writing is the candidate's own work because I have had him as a student for 4 years and know his work well. I also saw him working on it. He has a remarkable talent for computer science and mathematics.

This declaration must be signed by the supervisor; otherwise a grade may not be issued.

I have read the final version of the extended essay that will be submitted to the examiner.

To the best of my knowledge, the extended essay is the authentic work of the candidate.

I spent [/] hours with the candidate discussing the progress of the extended essay.

Supervisor's signature: Date:

Assessment form (for examiner use only)

Criteria	Achievement level				
	Examiner 1	maximum	Examiner 2	maximum	Examiner 3
A research question	1	2		2	
B introduction	1	2		2	
C investigation	2	4		4	
D knowledge and understanding	2	4		4	
E reasoned argument	2	4		4	
F analysis and evaluation	2	4		4	
G use of subject language	3	4		4	
H conclusion	1	2		2	
I formal presentation	2	4		4	
J abstract	1	2		2	
K holistic judgment	2	4		4	
Total out of 36	19				

Should computer processor development focus be shifted from sequential to parallel computation?

Computer Science

Session: May 2013

Submitted in partial fulfillment of the requirements of the International Baccalaureate Diploma

Word Count: 3964

Abstract

Should computer processor development focus be shifted from sequential to parallel computation? This question asks whether or not the resources allocated to technological development should be shifted from a sequential computation based focus to a parallel computation based focus. Moore's law, which states that the density of a computer chip doubles every 18 months, is quickly reaching the end of its validity. A transistor can only be so small, and they are quickly reaching that limit. The solution to this problem may be found in parallel processing. This paper gives a brief introduction to the general theory of parallel computation and its limits and capabilities, followed by an investigative example of parallel programming to reaffirm theory in real world practice.

The investigation led to the conclusion that the advantages of parallel computation outweigh its disadvantages and therefore it is worthwhile to shift focus from sequential computation to parallel computation. More parallel algorithms and more efficient parallel systems should be developed in order to ensure the continuation of technological improvement.

Word Count: 169

Table of Contents

Introduction

Should processor development focus be shifted from sequential to parallel?

Moore's Law states that the processing power of a computer doubles every 18 months (Kaku). However, processors are beginning to reach their physical limits and the growth of processing power has slowed (Kaku). There is already a trend to sidestep this slowing growth by increasing the number of cores per CPU, but the technology is not used to its maximum potential, due to slow growth in the parallel field relative to that of the sequential. Most programming languages have been designed with sequential computation in mind, and although a few have parallel APIs, they are nowhere near as refined as their sequential counterparts (Consider the standard for-loop, a clean, efficient sequential coding practice, compared to Java's Thread object, with its multiple auxiliary classes and lack of organization). Thus, an increase in parallel processing research may make more use of the growing trend to produce parallel hardware, further advance the production of such hardware, and pose as a potential solution to nature's physical limits on sequential processing. However, although the intuitive thought would be to simply continue to multiply the number of processors in use in order to increase computer speed, but that may not always be the case.

Parallel processing algorithms split otherwise sequential processing algorithms into various chunks that can be run simultaneously. Unfortunately, because life is not ideal, distributing processing requires a certain amount of processing in itself, or overhead, which may make parallel processing inefficient in certain cases.

The purpose of this essay is to determine whether or not parallel processing's advantages outweigh its disadvantages enough for research into technological improvement to be shifted

away from sequential processing and towards parallel processing. This will be accomplished through an investigative approach, first describing parallelism itself, then the theory surrounding itself, followed by a real-world experiment.

Word Count: 307

Summary of Evidence

A. General

Parallel processing employs multiple processors to solve problems in a concurrent fashion rather than a sequential fashion. This is done by "breaking up a task and having processors work on the parts independently (Smith 3)." The reasoning behind this is that if you have a problem that takes t time to solve and you split it up into N portions that all run at the same time, it should theoretically take t/N time to solve this new problem.

There are several ways of going about solving a problem in a parallel manner, but all of them have one thing in common: they require multiple processors. Because processors can only complete one instruction, or process, at a time, having multiple processors that can all communicate with each other, either with shared memory or some other physical connection, allows for multiple processes to occur concurrently. These connections allow for processors to combine their output data into one final output for a given input. Some examples of parallel systems are SMP, or shared memory multiprocessor, cluster, and hybrid (Kaminsky 22). Shared memory multiprocessor computers are computers where each processor has its own CPU/Cache unit and shares a main memory with the others (Kaminsky 22). Cluster parallel computers have frontend processors to distribute the problem to several backend processors with their own CPU's and memories (Kaminsky 22). There is no main, shared, memory in cluster computers

(Kaminsky 22). Hybrid parallel computers are simply a combination of the two (Kaminsky 25). Neither of these systems is necessarily better than the other, but each is "best suited for certain kinds of problems" and each has it's own requirements and dependencies (Kaminsky 28). These are only some of the systems of parallel processing, and there are many more ways to construct a parallel system. The multitude of possible parallel processing systems means that parallelism has many more degrees of freedom than sequential computing when it comes to problem solving (Skillicorn 5). These degrees of freedom can make it difficult to find the optimal solution to a given problem when using a parallel construction (Skillicorn 5).

In order to go about solving a problem in a parallel fashion, some general steps are to be taken. First the problem must be analyzed and split into different processes that can be done concurrently. This is best shown through example of how pipelining, a popular parallel method that is even used in some high-end single-core processors, would be implemented. Consider the function $f(x, y) = 2(x + y)^2$. If you decompose this function using the mathematical order of operations, you have three steps. First the parenthesis must be calculated, so $(x + y)$, which, for the purposes of this example equals α. Then, the exponent must be solved, so α^2, which, equals β. Finally, the multiplication must be processed, so $2 * \beta$, which is the output of the function. Using this decomposition we can parallelize the problem. Given the input (x, y), we pass it to a processor n_1, which is dedicated solely to that function $f_a(x, y) = x + y$. When that processor completes the calculation, it passes its result to processor n_2, which computes $f_b(\alpha) = \alpha^2$, and then takes the next input. Finally, n_2 passes its results to n_3, which computes $f_c(\beta) = 2 * \beta$ and then outputs the result. This may seem like a waste of time, since three computations takes almost no time with the processing power of today's computers, but if thought of in time cycles and with large amounts of inputs, it certainly does. Assume that each instruction takes only one

cycle, regardless of the processor used or the type of instruction and ignoring input and output. On a sequential processing unit, for every input, it will take one cycle to add, one cycle to square, and one cycle to multiply, giving a total of three cycles per computation. This may not seem like much at all, but given 1 billion input numbers, it would take 3 billion cycles to output all of the solutions. Now, consider a pipelining processing system. The first processor would take once cycle to compute the addition. Then, after passing on the output, it would accept another input and begin processing it, therefore, in that one cycle, two outputs are produced: that of the first and second processor. After that cycle, one cycle would produce three outputs (one for each processor) and according to our example, one of those outputs would be our solution. Thus, after three cycles, the machine would begin to produce one output per cycle, whereas the sequential unit would produce one output for every three cycles. This reduction in time, while small, scales over expressions involving larger amounts of computations (so a five step computation would take $1/5^{th}$ the time to solve using pipelining) and is useful when large amounts of input are provided. However, there are also fallbacks to this method of parallelism. Firstly, it is very difficult to make a pipelining system adaptable, that is, you can set up an architecture to solve one function, but in order to change it to solve another, it will take some time and is tedious (Fountain 33). Furthermore, in reality, not all computations take the same amount of time, and thus a processor in the pipeline may bottleneck the entire process. This could be solved using multiple pipelining systems that have multiple processors per instruction, but this would simply add to the overhead and complexity of the solution, further slowing it down. Also, pipelining can only solve a very particular type of problem and does not work for all solutions (Fountain 33). Thus, in pipelining, and other parallel solutions, there are both pros and cons that must be weighed when deciding what to implement.

Thus, although there are many parallel computation models, it is difficult to choose which model is appropriate for a specific problem. Furthermore, some theorize that there may not be a correct parallel model for certain problems (Smith 4). These inherently sequential problems would have no parallel solution more optimal that its most effective sequential solution (Smith 4). The existence of such problems would surely be a huge blow for parallel computing, and many problems appear to be inherently sequential. However, none have been found and problems that have seemed inherently sequential simply needed to be tackled from another angle (Smith 4).

Word Count: 1041

B. Limits to Speedup and Sizeup

Parallelism undoubtedly speeds up certain problems, but there are restrictions. First, let us define speed up: If $N \equiv Problem\ Size$, $K \equiv Number\ of\ Processors$, and $T \equiv Time\ to\ Solve\ a\ Problem$, because T is dependent on several factors such as processor speed, problem size, and number of processors, we can define the amount of time to solve a problem as T (N, K) (Kaminsky 100). Furthermore, because we are considering two types of computing, we can split this up into $T_{seq}(N,K)$ and $T_{par}(N,K)$. Now, because speed is the reciprocal of running time, $S(N,K) = \frac{1}{T(N,K)}$ (Kaminsky 100). With this definition of speed, we can define speedup as the ratio of the speed of the parallel version to the speed of the sequential version (which is NOT the parallel version using one processor so as to not cancel out the overhead of the parallel version in this comparison) (Kaminsky 101). Thus, $Speedup(N,K) = \frac{S_{par}(N,K)}{S_{seq}(N,1)} = \frac{\frac{1}{T_{par}(N,K)}}{\frac{1}{T_{seq}(N,K)}} = \frac{T_{seq}(N,K)}{T_{par}(N,K)}$, which can be used to define the efficiency of a parallel program with respect to the number of processors and the sequential version as $Eff(N,K) = \frac{Speedup(N,K)}{K}$ (Kaminsky 101).

This efficiency should be equal to one in an ideal parallel program, because the speed up would increase by factors of K, which would then be divided by K (Kaminsky 101). However, because most programs are not ideal, this is not so.

Gene Amdahl described parallel computing as a sequential portion that every program must have, followed by the parallel portion (Kaminsky 102). Parallel programs have overhead that must be solved sequentially, such as start up and the distribution of processing requirements over the network. If this sequential fraction F is incorporated into the T (N, K) function, we arrive at $T(N,K) = F * T(N,1) + \frac{1}{K}(1-F) * T(N,1)$ (Kaminsky 103). Plugging this into our previous functions, we arrive at new speedup and efficiency functions:

$$Speedup(N,K) = \frac{T(N,1)}{T(N,K)} = \frac{T(N,1)}{F * T(N,1) + \frac{1}{K}(1-F) * T(N,1)} = \frac{1}{F + \frac{1-F}{K}}$$

$$Eff(N,K) = \frac{Speedup(N,K)}{K} = \frac{1}{KF + 1 - F}$$

(Kaminsky 103).

Now, we find that as the limit of K goes to infinity in Speedup (N,K), the speedup approaches 1/F, those posing as a limit for the speedup of a sequential problem using parallel means (Kaminsky 103). Furthermore, as the limit of K goes to infinity in Efficiency (N,K), the efficiency approaches 0 (Kaminsky 103). Therefore, adding processors does not exactly entail an increase in efficiency. The larger the sequential portion, the more quickly a function becomes completely inefficient. These restrictions imply that parallelism is useful only when the sequential fraction can be kept small, a difficult feat in itself. Consciousness itself is sequential, although it may seem parallel, and therefore coming up with the proper parallel solution with a small sequential fraction can be difficult (Skillicorn 5). Even in the aforementioned pipelining

example, the sequential fraction would be quite large in actual implementation, especially if attempting to make a more generic pipelining system.

Another method of determining parallel-sequential trade off is through sizeup. Rather than "expressing a program's running time as a function of the problem size and the number of processors… we can just as easily turn that relationship around and express the program's problem size as a function of the running time and the number of processors" and thus consider sizeup rather than speedup (Kaminsky 122). First, let us define the program's problem size as $N\ (T, K)$ and its size-up as $Sizeup(T, K) = \frac{N_{par}(T,K)}{N_{seq}(T,K)}$ which implies that, ideally, a system with K processors should be able to solve K times more problems (Kaminsky 122). The efficiency of such a system with respect to sizeup is $SizeupEff(T, K) = \frac{Sizeup(T,K)}{k}$ (Kaminsky 122). John Gustafson argued that sizeup was more practical than speedup and that when scaling up a problem, the parallel fraction scales up, not the sequential fraction (Kaminsky 123). Now, incorporating the sequential fraction into sizeup we arrive at:

First we assume that the program's sequential portion is independent of N and parallel portion running time is directly proportional to N:

$$T(N, K) = a + \frac{1}{K} dN$$

Solving for N:

$$N(T, K) = \frac{1}{d} K(T - a)$$

$$Sizeup(T, K) = \frac{N(T, K)}{N(T, 1)} = K$$

$$SizeupEff(T, K) = \frac{Sizeup(T, K)}{K} = 1$$

(Kaminsky 125).

Therefore, programs experience ideal sizeup when increasing the number of processors if the sequential portion does not scale up with the number of problems (Kaminsky 125). However, this is only an approximation, as the sequential portion's running time does increase with the number of problems (Kaminsky 125). This is shown through the following equations:

First, we define (a + bN) as the running time for the sequential portion and (c+dN) as the running time for the parallel portion. So:

$$T(N,K) = (a + bN) + \frac{1}{K}(c + dN)$$

Solving for N:

$$N(T,K) = \frac{T - a - \frac{c}{K}}{b + \frac{d}{K}} = \frac{KT - Ka - c}{Kb + d}$$

$$Sizeup(T,K) = \frac{N(T,K)}{N(T,1)} = \frac{(KT - Ka - c)(b + d)}{(T - a - c)(b + dK)}$$

a and c are both small compared to the other factors, so they can be eliminated:

$$Sizeup(T,K) = \frac{(KT)(b + d)}{(T)(Kb + d)} = \frac{Kb + Kd}{Kb + d} = \frac{K\left(\frac{b}{d}\right) + K}{K\left(\frac{b}{d}\right) + 1}$$

$$SizeupEff(T,K) = \frac{Sizeup(T,K)}{K} = \frac{\frac{b}{d} + 1}{K\left(\frac{b}{d}\right) + 1}$$

(Kaminsky 126).

These alterations of the sizeup law change the entire nature of sizeup. Now, rather than never reaching a point of inefficiency, if we take the limit as the number of processors approaches infinity we get zero. Therefore, although sizeup is more practical that speedup, it does also have an inherent limit, according to Gustafson (Kaminsky 126). Thus, parallel computation has a limit to its efficiency.

Word Count: 799

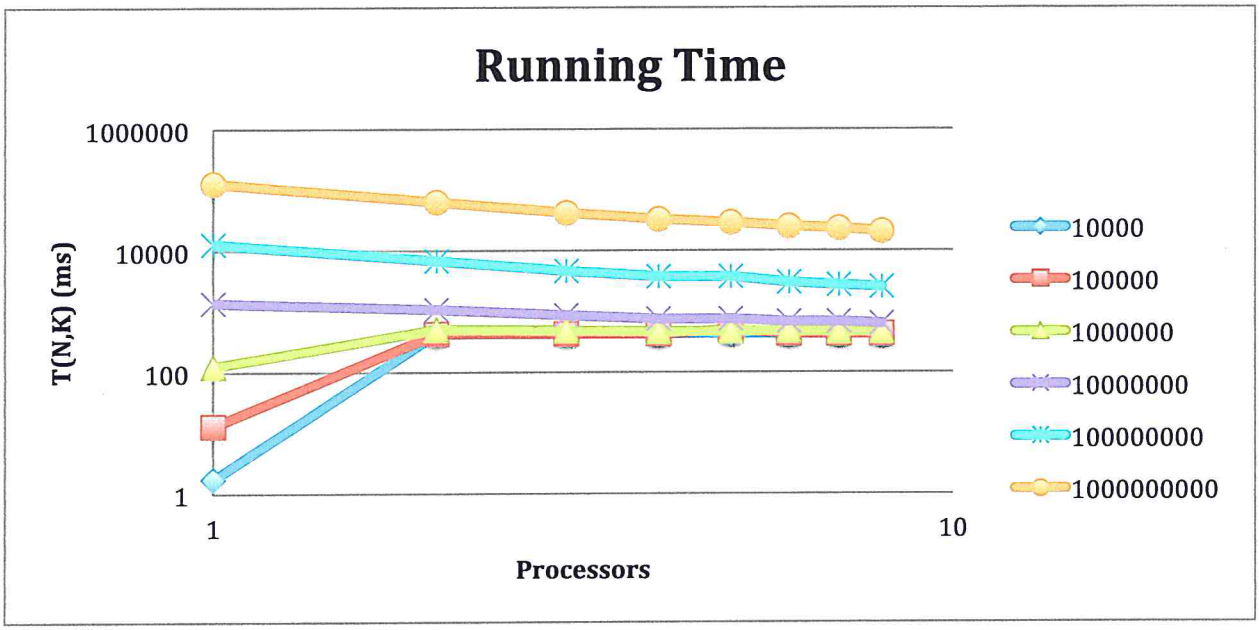
C. Monte Carlo Investigation

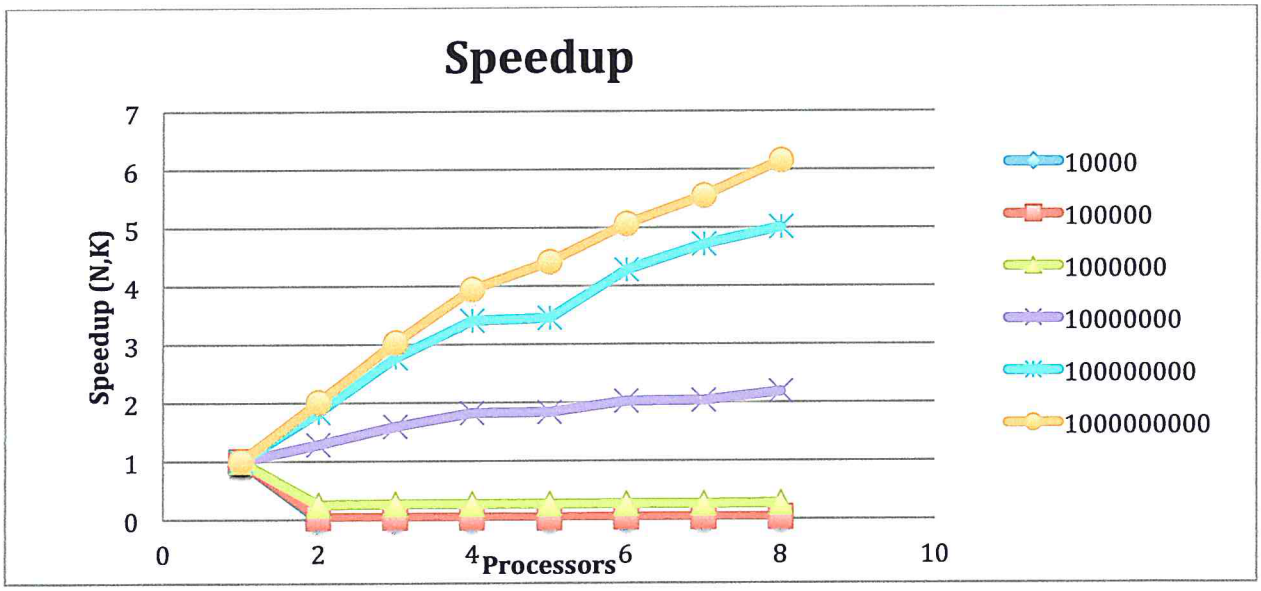
In order to display these effects of the sequential portion of a parallel program, I have devised my own parallel program and its sequential counter part. The mathematical constant Pi can be quite accurately estimated using a Monte Carlo Algorithm (Kaminsky 168). The basic idea behind this algorithm is that of a dartboard. Imagine a dartboard of radius one, centered about the origin of a Cartesian plane. The area of this circle is $A = \pi r^2 = \pi 1^2 = \pi$, and if you split it into four parts, its area is $\pi/4$. Now, the quadrant in which it lies has an area of $A = s^2 = 1^2 = 1$, thus the ratio of the area of the sector to the quadrant is $\frac{\frac{\pi}{4}}{1} = \frac{\pi}{4}$. If a large number of "darts" are thrown at this quadrant, the ratio of those landing within the sector to all of them is $\frac{\pi}{4}$. Thus, finding the ratio of darts landing within the sector to all of the darts thrown, and multiplying by four gives an approximation of Pi. This algorithm actually suits both a parallel and sequential program very well. In order to implement it sequentially, it actually only took a for-loop containing the random generation of an x and y coordinate less than 1 and finding if the distance is at most one from the origin, which, if true, added one to a counter. This can be summarized by the pseudo code below:

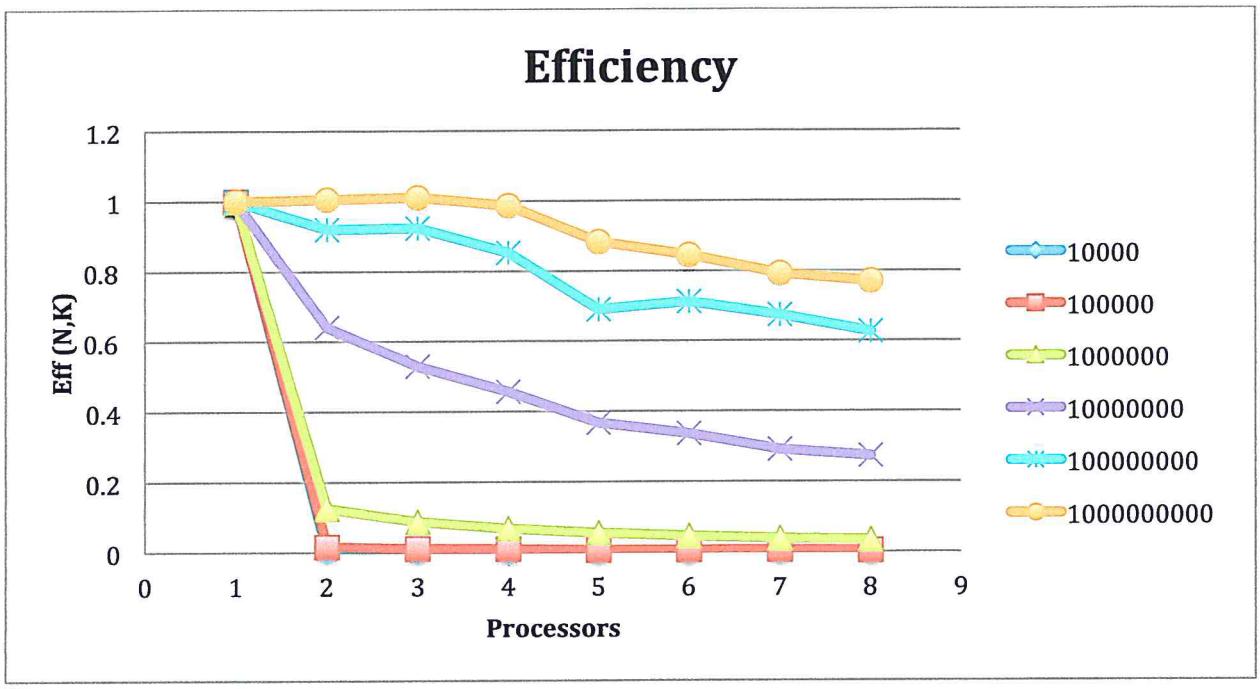
```
function montePi (throws) {
      count = 0; // amount of darts landing within the sector
      for (i = 0; i < throws; i++) { //iterate through dart throws
            x = random(); //generate random x-value less than 1
            y = random(); //generate random y-value less than 1
            if (x*x + y*y <= 1) { //distance formula
                  count++; //increment count
            }
      }
      return count/throws; //return ratio
}
```

As this code shows, the algorithm is very simple and very efficient, but, because of the nature of the problem, the for-loop must be iterated many times. Normally, this would not be a problem as sequential computers can compute simple algorithms as the one above fairly quickly, but when trying to get higher and higher degrees of accuracy, the time for completion takes substantially longer. In order to go about speeding up this process, I decided to create a temporary cluster parallel computer in the form of a local network. It is actually two programs. One program hosts a server and accepts client programs that will be doing the processing. The server divides the dart throws evenly and distributes the trial count to the clients and eventually records and combines their results. This can be considered the sequential fraction of the program, as it must be completed on one computer. In order to make this sequential fraction slightly more efficient, I used a "Thread" java object for accepting client connections. This object allows for "an application to have multiple threads of execution running concurrently" ("Thread") and therefore makes the sequential fraction smaller, since while the main thread is dividing the problem into smaller fractions, it does not have to waste time accepting client programs. The server computer had multiple processors and therefore each thread could be executed in a parallel fashion. In a sense, I added a parallel flare to the sequential fraction of the server. The clients simply compute the estimation using the given number of "dart throws" and send the estimation back to the server to combine and output. Using this system and the sequential version of the algorithm, which is simply the for-loop, given above, and some input-output, I will generate graphs of relevant data for several different system configurations (i.e. the number of processors) and numbers of "dart throws." In order to make comparisons accurate, I used the same model of computer for every processor used. Furthermore, the running time graph uses "log-log" axes because it is "better suited to plotting data that spans many orders of magnitude

(Kaminsky 108)."







These three graphs displaying the speedup information described in the previous section show that the sequential fraction of a parallel program plays a key role in its performance. For a relatively small program size (<10000000), parallelization was extremely inefficient. The sequential fraction took a certain amount of time, around 400 milliseconds per processor, and this process, as predicted by Amdahl's Law, limited the parallelization's effectiveness. Although for the larger problem sets there is a clear speed up, especially for 10 million dart throws, there is a clear curve to the downward trend of the efficiency, with the efficiency leveling off as the number of processors was increased. The larger problem sets did not show this, but that can be attributed to a smaller sequential fraction relative to the problem size.

Word Count: 712

Analysis

The disadvantages of parallel computation include difficulty, expense, and inefficiency in certain cases. For certain problems, it can be very difficult to come up with an optimal parallel solution. For example, the cumulative sum of an integer seems inherently sequential from a first glance (Smith 4). The cumulative sum of an integer is the sum of all the integers from 0 up to it and the sequential algorithm would be as easy as a recursive function adding up all the numbers sequentially. However, there is an optimal parallel solution, it is just much more complex and difficult to arrive to (Smith 4). There are many problems that are seemingly inherently sequential and make it difficult to implement parallelism. Furthermore, a proper parallel solution may be too specific to be useful, such as with pipelining. Another hindrance to the production of parallel solutions is the sequential nature of human consciousness. When implementing the Monte Carlo problem above, the solution was almost intuitively sequential and although easy to parallelize, it was much easier to implement sequentially. Parts of our mind do work in parallel, but the majority of our conscious lives takes place in a sequential fashion and thus inhibits us from forming parallel solutions (Skillicorn 5). However, just because something is difficult does not mean it is impossible and parallel solutions are very much achievable. Furthermore, parallel computation models are fairly expensive. A multiprocessor computer may not be much more expensive that a single processor variant, but super computers, which have hundreds, if not thousands, of processors, cost millions of dollars. This is not so accessible to the average user and could reduce the usefulness of increased research into parallel computation for the average computer user. Lastly, there are certain cases when parallelism is not so useful. For example, with the small problem sets for the Monte Carlo problem, the overhead of the parallel solution made the problem extraordinarily ineffective. The added overhead is small compared to the

processing times of billions of "dart throws", but large compared to a million or so "dart throws." This inefficiency makes a sequential solution a more optimal solution for small problem sets. Of course, the sequential fraction and overhead could have been reduced, but that would have been more difficult, and that practicality of such a task depends on whether implementation time or solution time is more valuable to the user. Thus, there are significant drawbacks to parallel computation. However, the pros to be introduced below help alleviate some of the problems stated above.

The advantages of parallel computation include an ease of implementation in problems belonging to a parallel domain (Skillicorn 4), its side-stepping of Moore's laws impending doom, and its cost effectiveness (Skillicorn 4). Some problems, having originated from the real world, where events occur in parallel, actually are easier to implement in a parallel fashion. For example, consider traffic lights, where some intersections use sensors rather than timers to determine when to switch the light from red to green. If one computer had to manage all the checking, processing, and changing involved with every intersection, things would update very slowly, as the processor would have to allocate some time to each intersection. Furthermore, the computer would have to handle all the exceptions, such as a police officer changing the color of the light in order to direct traffic manually after some accident. The computer would surely slow to a crawl after some time. However, if you use multiple computers, perhaps one for every traffic light, or a server of computers at the headquarters each given a specific intersection to control for some period of time and then handing it off to another computer, things would occur much more quickly and efficiently. That problem, having originated from a parallel world, lies in the parallel domain and is more easily solved in a parallel fashion. The time to solve this problem would be cut by a factor of the number of computers used. Aside from solving parallel problems with

much more ease, parallel computers would be able to solve the chaos ensuing the end of Moore's Law. If technology stops improving, the industry will collapse, as companies would have nothing left to develop. However, parallel computers can continue to grow and the systems can be continually improved upon. Also, parallel computers, although more expensive, are more cost effective. Although a super computer may cost millions of dollars, its processing power is unmatched by any sequential computer. It took 8 computers to substantially decrease the solution time of the Monte Carlo problem above, but the time that it saved may be more valuable than the cost. Imagine a problem that would take a thousand years to solve with a sequential computer. Assuming you can sale this down by one hundred with a one hundred-processor computer, the cost would be relatively low, considering the researchers would be able to live through the length of their program, rather than wait ten centuries. Also, as with all products, as more companies enter such a market, prices will invariably go down, making parallel computers more accessible to the average user as well as improve parallel systems which are too specific, such as pipelining. Lastly, the most important advantage is that any parallel computer can process a sequential algorithm, although slightly slower, whereas the same is not true in reverse.

Word Count: 893

Conclusion

The information presented in this paper gives evidence to support the notion that parallel computation could replace sequential computation. Parallel computation allows for increased speed in solving problems of large magnitude. Although parallel algorithms do not scale exactly by a factor of the number of processors, the speedup is significant. Also, there are always ways to reduce the sequential portion and thus reduce the effect of Amdahl's law. With this ever increasing potential for speed increase, parallel computation provides a means to avoid the limit

that sequential computation will soon reach. Besides these valuable traits, parallel computation allows for a more cost effective method of solving large, time-consuming problems and so far has not been found to be inefficient for any type of problem (i.e. no inherently sequential problems have been found). Furthermore, although it is difficult to come up with parallel solutions in certain cases, it is certainly not impossible and the advantages outweigh the added effort and cost. These advantages mean that there is enough reason to push for a change in focus from sequential to parallel computation. A larger focus should be held within the world of academia to the study of parallel algorithms and a large amount of research should be devoted to the development of parallel systems in the industrial sphere.

Word Count: 212

Works Cited

Akl, Selim G. *Parallel Sorting Algorithms.* Orlando, FL: Academic Press, Inc., 1985. Print.

Fountain, T.J. *Parallel Computing: principles and practice.*" New York, NY: Cambridge University Press, 1994. Print.

Heineman, George T. "Multi-threaded algorithm implementations." O'Reilly Media, Inc., 18 June 2009. Web. 23 October 2012.

"Thread." *Java™ Platform, Standard Edition 7 API Specification.* Oracle, 2012. Web. 02 Dec. 2012.

Kaku, Michio. "Tweaking Moore's Law: Computers of the Post-Silicon Era." *Big Think.* Big Think, Inc., 7 March 2012. Web. 23 September 2012.

Kaminsky, Alan. *Building Parallel Programs.* Boston, MA: Cengage Learning, 2010. Print

Skillicorn, David. *Foundations of Parallel Programming.* New York, NY: Cambridge University Press, 1994. Print.

Smith, Justin R. *The Design and Analysis of Parallel Algorithms.* New York, NY: Oxford University Press, Inc., 1993. Print.

The code for the Monte Carlo Investigation will be provided here. The program is written in Java.

```
/**
 * The sequential application
 * @author Eric Ponce
 */
public class MonteCarlo {

    /**
     * Will accept parameters for dart throws and trial counts.
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int dartCount, trialCount = 0;
        try {
            dartCount = Integer.parseInt(args[0]);
            trialCount = Integer.parseInt(args[1]);
        } catch (Exception e) {
            Scanner sc = new Scanner(System.in);
            System.out.println("Arguments could not be read.");
            System.out.print("Dart Count: ");
            dartCount = sc.nextInt();
            System.out.print("Trial Count: ");
            trialCount = sc.nextInt();
        }
        long totalTime = 0; //Timer code
        double estimationTotal = 0;
        for (int a = 0; a < trialCount; a++) {
            long time = System.currentTimeMillis();
            double countWithin = 0;
            for (int i = 0; i < dartCount; i++) {
                countWithin += (dartThrow()) ? 1 : 0;
            }

            estimationTotal += countWithin/dartCount;
            long newTime = System.currentTimeMillis() - time;
            totalTime += newTime;
        }
        System.out.println("Value: " + estimationTotal/trialCount*4);
        System.out.println("Total Time: " + totalTime + "ms.");
        System.out.println("Average Time: " + totalTime/trialCount + "ms.");
    }

    public static boolean dartThrow() {
        double x = Math.random();
        double y = Math.random();
        double result = x * x + y * y;
        return result <= 1.0;
    }
}
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.Scanner;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * The server for the parallel program
 * @author Eric Ponce
 */
public class MonteCarloPiServer {

    static ArrayList<Socket> clients = new ArrayList<Socket>();

      public static void main(String[] args) throws IOException {
        final ServerSocket serverSocket = new ServerSocket(4444);
        Scanner sc = new Scanner(System.in);
        System.out.println("Socket Opened. Accepting Connections");
        new Thread(new Runnable() {
            @Override
            public void run() {
                while (true) {
                    try {
                        Socket client = serverSocket.accept();
                        clients.add(client);
                        System.out.println("Client Added");
                    } catch (IOException ex) {
                        System.out.println("Error. Exiting.");
                        System.exit(0);
                    }

                }
            }
        }).start();
        while (sc.hasNext()) {
            String s = sc.nextLine();
            String[] parts = s.split(" ");
            int dartThrows = Integer.parseInt(parts[0]);
            int trialCount = Integer.parseInt(parts[1]);
            if (parts[2].equalsIgnoreCase("BEGIN")) {
                ArrayList<BufferedReader> readers = new
ArrayList<BufferedReader>();
                ArrayList<PrintStream> printers = new
ArrayList<PrintStream>();
                for (Socket c : clients) {
                    readers.add(new BufferedReader(new
InputStreamReader(c.getInputStream())));
                    printers.add(new PrintStream(c.getOutputStream()));
```

```
                }
                int n = clients.size();
                int trialsPerClient = dartThrows / n;
                double value = 0;
                double estimationTotal = 0;
                long timeTotal = 0;
                for (int i = 0; i < trialCount; i++) {
                    long timeRunning = System.currentTimeMillis();
                    for (PrintStream out : printers) {
                        out.println(trialsPerClient);
                    }
                    int inputsRecieved = 0;
                    while (inputsRecieved != n) {
                        for (BufferedReader in : readers) {
                            if (in.ready()) {
                                value += Double.parseDouble(in.readLine());
                                inputsRecieved++;
                            }
                        }
                    }
                    timeRunning = System.currentTimeMillis() - timeRunning;
                    value /= n;
                    estimationTotal += value;
                    value = 0;
                    timeTotal += timeRunning;
                }
                System.out.println("Value Estimated: " + (estimationTotal /
trialCount));
                System.out.println("Took a total time of " + timeTotal + "ms.
and an average time of " + (timeTotal / trialCount) + "ms.");
            }
        }
    }
}
```

```
import java.io.BufferedReader;
```

```
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Scanner;
/**
```

```
 * The client portion of the parallel program
 * @author Eric Ponce
 */
public class MonteCarloPiClient {
      public static void main(String[] args) throws UnknownHostException,
IOException {
        String ip = "";
        try {
            ip = args[0];
        } catch (Exception e) {
            System.out.print("Argument could not be read. ");
            Scanner sc = new Scanner(System.in);
            System.out.print("Enter IP Address: ");
            ip = sc.nextLine();
            sc.close();
        }
        final Socket socket = new Socket(ip, 4444);
        PrintStream p = new PrintStream(socket.getOutputStream());
        BufferedReader b = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
        System.out.println("Connected to Server");
        String input;
        while ((input = b.readLine()) != null) {
            System.out.println("Input Received. Calculating dart throws");
            int n = Integer.parseInt(input);
            double value = 0;
            for (int i = 0; i < n; i++) {
                double x = Math.random(); double y = Math.random();
                value += (x * x + y * y <= 1) ? 1 : 0;
            }
            value = value / n * 4;
            System.out.println("Calculated " + n + " dart throws and returned
" + value + ".");
        }
        p.close();b.close();socket.close();
    }
}
```

MonteCarlo Trial Data

Trials	Dart Throws	Total Time	Average Time	Estimation	Clients	Parallel?
100	10000	172	1.72	3.144424	1	N
100	10000	42229	422	3.143772	2	Y
100	10000	42337	423	3.14481848	3	Y
100	10000	42245	422	3.143468	4	Y
25	10000	10918	436	3.14784	5	Y
25	10000	10686	427	3.14152861	6	Y
25	10000	11029	441	3.14180072	7	Y
25	10000	10764	430	3.140591	8	Y
100	100000	1310	13	3.141396	1	N
50	100000	21279	425	3.1406952	2	Y
50	100000	21387	427	3.1408746	3	Y
100	100000	42198	421	3.1417952	4	Y
25	100000	12213	488	3.1413808	5	Y
25	100000	10763	430	3.1410888	6	Y
25	100000	10796	431	3.1423459	7	Y
25	100000	10686	427	3.1419409	8	Y
100	1000000	12496	124	3.1417634	1	N
25	1000000	12480	499	3.14208896	2	Y
25	1000000	11981	479	3.14124298	3	Y
100	1000000	46363	463	3.14159656	4	Y
25	1000000	11636	465	3.14111552	5	Y
25	1000000	11669	466	3.14111256	6	Y
25	1000000	11544	461	3.14146682	7	Y
25	1000000	11342	453	3.1411134	8	Y
100	10000000	136208	1362	3.141658	1	N
25	10000000	26615	1064	3.1417121	2	Y
25	10000000	21153	856	3.14169863	3	Y
100	10000000	74630	746	3.1415871	4	Y
25	10000000	18546	741	3.14152	5	Y
25	10000000	16878	675	3.14154082	6	Y
25	10000000	16755	670	3.14149125	7	Y
25	10000000	15601	624	3.14163238	8	Y
10	100000000	125848	12584	3.14152606	1	N
20	100000000	136807	6840	3.14159395	2	Y
20	100000000	90839	4541	3.14157433	3	Y
100	100000000	369438	3694	3.14159075	4	Y
25	100000000	91153	3646	3.14157886	5	Y
25	100000000	73675	2947	3.1416048	6	Y
25	100000000	66722	2668	3.141626	7	Y
25	100000000	62818	2512	3.14159842	8	Y
5	1000000000	639958	127991	3.14156738	1	N
5	1000000000	318206	63641	3.14154676	2	Y
5	1000000000	210946	42189	3.14157228	3	Y
10	1000000000	324198	32419	3.14160028	4	Y
10	1000000000	289806	28980	3.14159671	5	Y
10	1000000000	252794	25279	3.14158751	6	Y
10	1000000000	230802	23080	3.14157692	7	Y
10	1000000000	207993	20799	3.1416198	8	Y